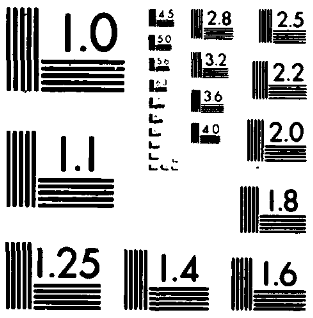
AD-A093 441   WHARTON SCHOOL PHILADELPHIA PA DEPT OF DECISION SCIENCES   F/G 9/2
THE USE OF DATA TYPE INFORMATION IN AN INTERACTIVE DATABASE ENV--ETC(U)
NOV 80 P BUNEMAN, I WINSTON   N00014-75-C-0462
UNCLASSIFIED 80-11-08   NL

END
DATE FILMED
1 81
DTIC


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

AD A093441

READ INSTRUCTIONS BEFORE COMPLETING FORM

REPORT DOCUMENTATION PAGE

14

1. REPORT NUMBER 80-11-08	2. GOVT ACCESSION NO. AD A093411	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 THE USE OF DATA TYPE INFORMATION IN AN INTERACTIVE DATABASE ENVIRONMENT.		5. TYPE OF REPORT & PERIOD COVERED technical/ 4/80-4/81
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Peter Buneman, Ira Winston	1	8. CONTRACT OR GRANT NUMBER(s) 15 N00014-75-C-0462
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Decision Sciences The Wharton School University of Pennsylvania, Phila., PA 19104		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Task NR049-272
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research	11	12. REPORT DATE ~~November 1980~~ Nov 80
9 Technical rept. Apr 80 - Apr 81		13. NUMBER OF PAGES 9
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) 12 11	LEVEL	15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Distribution unlimited

DTIC ELECTE JAN 5 1981 D

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Data type information, interactive database environment; data models; programming languages, databases and artificial intelligence; database schema, relational, end-users, designers, administrators, applications programmers,

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Despite the enormous advances that have been made in the specification of data types and data models in the fields of programming languages, databases and artificial intelligence; there remain a number of problems in attempting to unify the various approaches to the formal description of data. The purpose of this brief paper is to examine these problems from the point (or points) of view of those people - designers, administrators, applications programmers, and end-users - whose main interest is with databases. In

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

408757

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

80 12 30 006

20. (cont'd)

particular, we hope to display special concern for the tools provided for the end-user, who should be the final beneficiary of whatever advances are made.

The Use of Data Type Information in an Interactive Database Environment

Peter Buneman

Ira Winston

Department of Computer and Information Science

University of Pennsylvania, Pa 19104

Despite the enormous advances that have been made in the specification of data types and data models in the fields of programming languages, databases and artificial intelligence; there remain a number of problems in attempting to unify the various approaches to the formal description of data. The purpose of this brief paper is to examine these problems from the point (or points) of view of those people -- designers, administrators, applications programmers, and end-users -- whose main interest is with databases. In particular, we hope to display special concern for the tools provided for the end-user, who should be the final beneficiary of whatever advances are made.

In order to pin down some of these problems, it is worthwhile to attempt a definition of certain terms used in databases:

This work was partly supported by the Office of Naval Research under contract N00014-75-C-0462, and by a grant from the Digital Equipment Corporation.

1. A data model (or database management system if one is describing an implementation) is a set of parameterized or "generic" data types.

2. A database schema is a set of data types that result from instantiating the generic types of the data model to produce a set of data types that describe the data to be stored.

3. A database is an instantiation of those types defined by a schema.

For example, the terms "domain" and "relation" refer to the generic types that define the relational model. To create a specific relational schema involves instantiating domains: EMP#, NAME, SAL and relations, EMPLOYEE(EMP#, NAME, SAL...). A specific relational database is produced in turn by instantiating, for example the EMPLOYEE relation. There are of course, some flaws with these definitions; and it will be noticed that we have really punted on the problem of providing a definition of database terms to those concerned with the formalization of type definitions. However, the correspondence is useful in that it will allow us to draw some important contrasts between the use of databases and other forms of programming.

In a conventional programming task it is usually the case that one person, or a small group of people, takes responsibility for the design and implementation of a program. Each person involved would have a more or less complete understanding of the three levels of type definition described above. The situation is very different in a database environment. The people that implement the generic types (the database management system implementors) usually have no contact with the those who instantiate these to produce the specific types (the database designers) and these in turn have little to do with the end-users and applications programmers. This is a traditional distinction in roles; and it is to be hoped that, with increasingly powerful software, some of the distinctions will die out. It is unlikely, though, that an end user would normally want to implement a database management system. What we shall ask is what kinds of programming tools are appropriate for these programming environments whose differences are characterized by the use of type information.

Of the highest level, that of specifying generic types, we shall have little to say except to remark that it is not clear that the generic type facilities provided by the recent "higher order" programming languages are powerful enough to describe accurately the abstract structures provided by a DBMS. The reader interested in this problem may care to try specifying in ADA [1], say, the generic type RELATION as it is used in Pascal-R [2]. Apart from the obvious syntactic difficulties, there is a semantic problem in adequately specifying how RELATION is to be parameterized -- presumably by a set of domains. Another problem lies in the degree to which type information should be available at "run-time": we encountered this in trying [3] to embed Codasyl [4] structures in ADA as generic types.

At the other end of our scale lie the "end-users" and "applications-programmers" the distinction between these two classes often results more from the programming tools they are given than what they accomplish. Someone who generates a report through the use of a simple query language is an end user while someone who generates the same report with a conventional programming language is an applications programmer. We feel very strongly that the fact that the end-user cannot often perform some simple and useful computations, such as adding two numbers, is a testimony to the inadequacy of his query language rather than to the intelligence of the applications programmer. Ideally, an end-user should have at his disposal both the benefits of a simple query language and a high-level programming language.

A number of programming systems have recently appeared that are either extensions of an existing languages [2] that include data types appropriate to the implementation of a specific data model, or are new programming languages with an integral data management system [5,6]. These constitute an enormous advance over what has been previously available for programmers, which often consisted low-level access to some DBMS without the benefit of any structured program control. However, there are certain respects in which a data model cannot sit comfortably in any conventional programming language. The reason is that databases contain a high degree of regularity and may be conveniently manipulated by a set of "bulk" operators. The relational calculus [7] is a prime example of such operators, but they are also implicit in some of the non-relational query languages and are, of course, well known in the (unconventional) languages LISP and APL. To take an example of the incompatibilities that may arise when a relational database system is embedded in a conventional programming language, consider the problem of adding two columns of a relation row by row to produce a new relation with one fewer columns. This cannot be done within the relational calculus (although some query languages allow it), and a programmer using a relational database embedded in a programming language may find that an iterative program with low-level data access is required to perform this operation while an apparently more complex operation (such as a relational join) is available as a primitive operator.

An alternative approach is to take a set of database operators and to extend these to a full programming language. We have been experimenting with such a system over the past year or so that is based upon the Functional Query Language (FQL) [8]. It exploits a functional model of data: a database consists of a set of extensionally defined functions and contains a small set of functionals for operating either on data or user defined functions. Examples of these operators are <u>extension</u>, which applies a given function to a set or sequence. This is the familiar MAPCAR of LISP and is implicit in many APL functions. It also has close ties with the relational projection operator. Another example is <u>restriction</u>, which filters out the members of a sequence or set that do not satisfy a predicate; it has a direct counterpart in the relational calculus. Our purpose is not to go into the details of the language, but to indicate some of the advantages that the surrounding system may have for a wide variety of users.

1. It is an interactive system that contains an editor and interpreter. The user does not constantly have to switch between programs to debug his definitions.

2. It exploits the idea of a workspace similar to that of APL. Contained in a workspace is both a database and a set of defined functions. In one sense it solves the problem of "persistent data" by making both programs and data equally persistent.

3. Data may be incrementally defined: there is no predefined schema. It is however possible to load a Codasyl database for the purpose of query.

As examples of the way definitions are typed in this system, the following are possible definition and declaration headings:

```
Def AVERAGE: *NUM -> NUM = ...
  (The generic operator * stands for
   "sequence of")

Def AVE_SAL: *EMPLOYEE -> NUM = ...

Def MANAGES: [EMPLOYEE, EMPLOYEE]
                    -> BOOLEAN = ...

Def CADR: *?X -> ?X = ...
  (?X is a type parameter)
```

```
Def PAIRS: [*?X, ?Y] -> *[?X,?Y] = ...


Def SORT: [[?X, ?X]->BOOL, *?X]
                           -> *?X = ...


Dcl NEWGRADE: EMPLOYEE -> CHAR = ...
```

The last of these shows the declaration of an extensionally defined function (i.e. data). This statement calls for the evaluation of its body and creates an updateable function. It is through declarations like this that the user may build his own database or increment an existing database to which he has access. This is a part of the system that we are currently developing.

These examples are similar to the typing facilities available in a number of applicative languages. The reason they are necessary is not for efficiency: the cost of run-time checking is insignificant when compared to the cost of i/o in conventional database systems! The reason is that certain database management systems demand a form of "compilation". The token 'EMPLOYEE' cannot be directly used to access a class of records; instead it must first be converted into some internal referent. It is sometimes impossible and frequently inefficient to perform this conversion at run-time. It should also be noted that these examples could also have been typed automatically; and we hope to incorporate an algorithm such as that suggested in [9] in the near future. Explicitly defining a type is, however, often a great help in formulating a complicated function definition.

A serious problem remains. At present a form of compilation takes place whenever a top-level expression is evaluated. All the definitions relevant to the evaluation of the expression are gathered together, their types checked and the necessary conversions performed. Thus the user is only made aware of certain type conflicts at "run-time". We believe that a better system would comment or grumble about a user's definitions as he types them in. But it is not easy to see how such a system could type a definition when many of the referents in that definition are undefined; moreover there are a number of problems with automatic typing algorithms when applied to the definitions of higher order functionals (such as extension). In general, there appear to be severe difficulties in obtaining the benefits of an interactive programming system and one which properly exploits a rich set of type constructs. A *solution to these problems* would be of general benefit, but would especially help in the design of interactive database interfaces.

ah, J. et al, "Rationale for the Design of ADA," ACM
AN notices, June 1979.

dt, J., "Some High Level Language Constructs for Data
pe Relation," ACM TODS, 2, 3, pp. 247-261, September

ıan, O. P., Root, D. J., and Menten L., "A CODASYL
face for PASCAL and ADA," Moore School Report,
rsity of Pennsylvania, August 1980.

YL Data Base Task Group, April 1971 Report.

rman, A. et al, "Report on the Programming Language
," TR-34, U. C., San Francisco, 1978.

, L., and Shoens K., "Data Abstraction, Views and
tes in RIGEL," Proceedings ACM SIGMOD, May 1979.

, E. F., "A Relational Model for Large Shared Data
s," Comm. ACM, Vol. 13, No. 6, pp. 377-387, June 1970.

nan, O. P., and Frankel, R. E., "FQL -- A Functional
 Language," Proceedings ACM SIGMOD, May 1979.

r, R. "A Theory of Type Polymorphism in
ramming," J. Computer and System Sciences, Vol 17,
375, 1978.